class Direction:  
  
 OLD\_TO\_NEW = 1  
 NEW\_TO\_OLD = 0  
  
  
class CONST:  
  
 def \_\_init\_\_(self, MAX\_M, MAX\_C, CAP\_BOAT, MAX\_TIME\_S, MAX\_NODES):  
 self.MAX\_M = MAX\_M  
 self.MAX\_C = MAX\_C  
 self.CAP\_BOAT = CAP\_BOAT  
  
 self.MAX\_TIME = MAX\_TIME\_S  
 self.MAX\_NODES = MAX\_NODES  
  
# TERMINAL\_STATE = State(-1, -1, Direction.NEW\_TO\_OLD, -1, -1, 0)  
# INITIAL\_STATE = None  
# # State(MAX\_M, MAX\_C, Direction.OLD\_TO\_NEW, 0, 0,0)

from Constants import Direction  
MAX\_M = 30  
MAX\_C = 30  
CAP\_BOAT = 20  
CNST = None  
  
class State(object):  
  
 def \_\_init\_\_(self, missionaries, cannibals, dir, missionariesPassed, cannibalsPassed, level, CONSTS, moves):  
 self.missionaries = missionaries  
 self.cannibals = cannibals  
 self.dir = dir  
 self.action = ""  
 self.level = level  
 self.missionariesPassed = missionariesPassed  
 self.cannibalsPassed = cannibalsPassed  
 self.CONSTANTS = CONSTS  
 self.moves = moves  
 global MAX\_M  
 global MAX\_C  
 global CAP\_BOAT  
 global CNST  
 if not CONSTS is None:  
 CNST = CONSTS  
 MAX\_M = CONSTS.MAX\_M  
 MAX\_C = CONSTS.MAX\_C  
 CAP\_BOAT = CONSTS.CAP\_BOAT  
  
 # pass True to count forward  
 def successors(self):  
 listChild = []  
 if not self.isValid() or self.isGoalState():  
 return listChild  
 if self.dir == Direction.OLD\_TO\_NEW:  
 sgn = -1  
 direction = "from the original shore to the new shore"  
 else:  
 sgn = 1  
 direction = "back from the new shore to the original shore"  
 for i in self.moves:  
 (m, c) = i  
 self.addValidSuccessors(listChild, m, c, sgn, direction)  
 return listChild  
  
 def addValidSuccessors(self, listChild, m, c, sgn, direction):  
 newState = State(self.missionaries + sgn \* m, self.cannibals + sgn \* c, self.dir + sgn \* 1,  
 self.missionariesPassed - sgn \* m, self.cannibalsPassed - sgn \* c, self.level + 1,  
 self.CONSTANTS,self.moves)  
 if newState.isValid():  
 newState.action = " take %d missionaries and %d cannibals %s." % (m, c, direction)  
 listChild.append(newState)  
  
 def isValid(self):  
 # obvious  
 if self.missionaries < 0 or self.cannibals < 0 or self.missionaries > MAX\_M or self.cannibals > MAX\_C or (  
 self.dir != 0 and self.dir != 1):  
 return False  
 # then check whether missionaries outnumbered by cannibals in any shore  
 if (self.cannibals > self.missionaries > 0) or (  
 self.cannibalsPassed > self.missionariesPassed > 0): # more cannibals then missionaries on original shore  
 return False  
 return True  
  
 def isGoalState(self):  
 return self.cannibals == 0 and self.missionaries == 0 and self.dir == Direction.NEW\_TO\_OLD  
  
 def \_\_repr\_\_(self):  
 return "\n%s\n\n< @Depth:%d State (%d, %d, %d, %d, %d) >" % (  
 self.action, self.level, self.missionaries, self.cannibals, self.dir, self.missionariesPassed,  
 self.cannibalsPassed)  
  
 def \_\_eq\_\_(self, other):  
 return self.missionaries == other.missionaries and self.cannibals == other.cannibals and self.dir == other.dir  
  
 def \_\_hash\_\_(self):  
 return hash((self.missionaries, self.cannibals, self.dir))  
  
 def \_\_ne\_\_(self, other):  
 return not (self == other)  
  
TERMINAL\_STATE = State(-1, -1, Direction.NEW\_TO\_OLD, -1, -1, 0, CNST,None)  
# INITIAL\_STATE = State(MAX\_M, MAX\_C, Direction.OLD\_TO\_NEW, 0, 0, 0, CNST)

from collections import defaultdict  
from State import TERMINAL\_STATE  
import time  
  
class Graph:  
  
 def \_\_init\_\_(self):  
 self.bfs\_parent = {}  
 self.dfs\_parent = {}  
 self.expandedBFS = 0  
 self.expandedDFS = 0  
  
 def BFS(self, s):  
 self.expandedBFS = 0  
 self.bfs\_parent[s] = None  
 visited = {(s.missionaries, s.cannibals, s.dir): True}  
 s.level = 0  
 start\_time = time.time()  
 queue = [s]  
 while queue:  
 self.expandedBFS += 1  
 u = queue.pop(0)  
 if u.isGoalState():  
 print("No of Expanded Nodes: " + str(self.expandedBFS))  
 print("No of Explored Nodes: " + str(visited.\_\_len\_\_()))  
 queue.clear()  
 self.bfs\_parent[TERMINAL\_STATE] = u  
 return self.bfs\_parent  
 # Stops searching after a certain time/node limit   
 t = time.time() - start\_time  
 if t > u.CONSTANTS.MAX\_TIME or self.expandedBFS > u.CONSTANTS.MAX\_NODES:  
 if t > u.CONSTANTS.MAX\_TIME:  
 print("%.2fs EXCEEDED TIME LIMIT of %.2fs" % (t, u.CONSTANTS.MAX\_TIME))  
 else:  
 print("EXCEEDED NODE LIMIT of %d" % u.CONSTANTS.MAX\_NODES)  
 print("No of Expanded Nodes: " + str(self.expandedBFS))  
 print("No of Explored Nodes: " + str(visited.\_\_len\_\_()))  
 queue.clear()  
 return {}  
 for v in reversed(u.successors()):  
 if (v.missionaries, v.cannibals, v.dir) not in visited.keys():  
 self.bfs\_parent[v] = u  
 v.level = u.level + 1  
 queue.append(v)  
 visited[(v.missionaries, v.cannibals, v.dir)] = True  
 return {}  
  
 def DFS(self, s):  
 self.expandedDFS = 0  
 self.dfs\_parent[s] = None  
 visited = {(s.missionaries, s.cannibals, s.dir): True}  
 start\_time = time.time()  
 stack = [s]  
 while stack:  
 u = stack.pop()  
 self.expandedDFS += 1  
 if u.isGoalState():  
 print("No of Expanded Nodes: " + str(self.expandedDFS))  
 print("No of Explored Nodes: " + str(visited.\_\_len\_\_()))  
 self.dfs\_parent[TERMINAL\_STATE] = u  
 stack.clear()  
 return self.dfs\_parent  
 t = time.time() - start\_time  
 # Stops searching after a certain time/node limit   
 if t > u.CONSTANTS.MAX\_TIME or self.expandedDFS > u.CONSTANTS.MAX\_NODES:  
 if t > u.CONSTANTS.MAX\_TIME:  
 print("%.2fs EXCEEDED TIME LIMIT of %.2fs" % (t, u.CONSTANTS.MAX\_TIME))  
 else:  
 print("EXCEEDED NODE LIMIT of %d" % u.CONSTANTS.MAX\_NODES)  
 print("No of Expanded Nodes: " + str(self.expandedDFS))  
 print("No of Explored Nodes: " + str(visited.\_\_len\_\_()))  
 stack.clear()  
 return {}  
 for v in u.successors():  
 if (v.missionaries, v.cannibals, v.dir) not in visited.keys():  
 visited[(v.missionaries, v.cannibals, v.dir)] = True  
 self.dfs\_parent[v] = u  
 stack.append(v)  
 return {}  
  
 # Prints the path returned by BFS/DFS  
 def printPath(self, parentList, tail):  
 if tail is None:  
 return  
 if parentList == {} or parentList is None: # tail not in parentList.keys():  
 return  
 if tail == TERMINAL\_STATE: tail = parentList[tail]  
 stack = []  
 while tail is not None:  
 stack.append(tail)  
 tail = parentList[tail]  
 while stack:  
 print(stack.pop())

#! /usr/bin/env python3  
import sys  
import time  
from Graph import Graph  
from State import State, Direction, TERMINAL\_STATE  
from Constants import CONST  
CON\_IN = sys.stdin  
CON\_OUT = sys.stdout  
  
# Generate All possible next moves for each state to reduce number of iterations on each node  
def genPossibleMoves(CAP\_BOAT):  
 moves = []  
 for m in range(CAP\_BOAT + 1):  
 for c in range(CAP\_BOAT + 1):  
 if 0 < m < c:  
 continue  
 if 1 <= m + c <= CAP\_BOAT:  
 moves.append((m, c))  
 return moves  
  
def runBFS(g, INITIAL\_STATE):  
 sys.stdout = open("outBFS.txt", "w")  
 print("\n\nBFS :: \n")  
 start\_time = time.time()  
 p = g.BFS(INITIAL\_STATE)  
 end\_time = time.time()  
 # print("Printing Solution...")  
 if len(p):  
 g.printPath(p, TERMINAL\_STATE)  
 else:  
 print("No Solution")  
 print("\n Elapsed time in BFS: %.2fms" % ((end\_time - start\_time)\*1000))  
  
def runDFS(g, INITIAL\_STATE):  
 sys.stdout = open("outDFS.txt", "w")  
 print("\n\nDFS :: \n")  
 start\_time = time.time()  
 p = g.DFS(INITIAL\_STATE)  
 end\_time = time.time()  
 if len(p):  
 g.printPath(p, TERMINAL\_STATE)  
 else:  
 print("No Solution")  
 print("\n Elapsed time in DFS: %.2fms" % ((end\_time - start\_time)\*1000))  
  
def main():  
 sys.stdin = open("in.txt", "r")  
 m = int(input("m="))  
 print(m, end="\n")  
 c = int(input("c="))  
 print(c, end="\n")  
 k = int(input("k="))  
 print(k, end="\n")  
 t = int(input("TIME\_LIMIT\_s="))  
 print(t, end="\n")  
 n = int(input("NODE\_LIMIT="))  
 print(n, end="\n")  
 CNST = CONST(m, c, k, t, n)  
 moves = genPossibleMoves(CNST.CAP\_BOAT)  
 print(str(moves.\_\_len\_\_())+" iterations per Node.")  
 INITIAL\_STATE = State(CNST.MAX\_M, CNST.MAX\_C, Direction.OLD\_TO\_NEW, 0, 0, 0, CNST, moves)  
 # TERMINAL\_STATE = State(-1, -1, Direction.NEW\_TO\_OLD, -1, -1, 0)  
 g = Graph()  
 sys.stdout = CON\_OUT  
 print("\nRunning BFS>")  
 runBFS(g, INITIAL\_STATE)  
 sys.stdout = CON\_OUT  
 print("Executed BFS>")  
 print("\nRunning DFS>")  
 runDFS(g, INITIAL\_STATE)  
 sys.stdout = CON\_OUT  
 print("Executed DFS>")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()